Syntax Analysis - Parsing

66.648 Compiler Design Lecture (02/02/98)

Computer Science

Rensselaer Polytechnic

Lecture Outline

- Bottom-up Parsing
- Top-down parsing
- Administration

Bottom-up Parsing

YACC uses bottom up parsing. There are two important operations that bottom-up parsers use. They are namely shift and reduce.

(In abstract terms, we do a simulation of a Push Down Automata as a finite state automata.)

Input: given string to be parsed and the set of productions.

Goal: Trace a rightmost derivation in reverse by starting with the input string and working backwards to the start symbol.

Algorithm

1. Start with an empty stack and a full input buffer.

(The string to be parsed is in the input buffer.)

2. Repeat

a. <u>Shift</u> zero or more input symbols onto the stack from input buffer until a handle (beta) is found on top of the stack. If no handle is found report syntax error and exit.

b. <u>Reduce</u> handle to the nonterminal A.

(There is a production A := beta)

until the input buffer is empty and the stack contains the start symbol.

Algorithm Cont...

3. Accept input string and return some representation of the derivation sequence found (e.g.., parse tree)

The four key operations in bottom-up parsing are shift, reduce, accept and error.

Bottom-up parsing is also referred to as shift-reduce parsing.

Important thing to note is to know when to shift and when to reduce and to which reduce.

Example of Bottom-up Parsing

STACK	INPUT BUFFER	ACTION
$	num1 +num2 * num3$	shift
$num1	+ num2 * num3$	reduc
$T	+ num2 *num3$	reduc
$E	+ num2 *num3$	shift
$E+	num2*num3$	shift
$E+num2	*num3$	reduc
$E+T	*num3$	shift

Bottom-up Parsing Example contd

E+T*	num3 $	shift
E+T*num3	$	reduc
E+T*F	$	reduc
E+T	$	reduc
E	$	accept

Top-Down Parsing

As against bottom-up parsing, top-down parsing finds a leftmost derivation starting with the start symbol. Top-down parsing is used in commercial compilers.

The algorithm that is commonly used as a top-down parser is a recursive-descent parser.

In order to use recursive descent parsing, the grammar should NOT be left recursive and no two right side of a production has a common grammar symbol.

Examples of Left Recursion

E ---> E + T | T.

Left recursion can be eliminated by

E--> TE'

E' --> +TE' | epsilon.

Left Recursion algorithm is described in algorithm 4.1 (page 177)

Left Factoring

S --> if E then S | if E then S else S

can be rewritten as

S --> if E then S S'

S' --> else S | epsilon

Left factoring algorithm is given in page 178.

Recursive Descent Parsing

parsing expressions: (page 75)

e --> term moreterms

moreterms --> + term moreterms | epsilon

term --> num

```
int lookahead;
parse()
{   lookahead=lexan() /* ge the next token */
    while (lookahead != DONE) { expr(); match(';');}
}
```

Recursive Descent Parsing cont...

```
expr() {
    int t;
    term();
    while(1)
    switch (lookahead) {
    case '+' :  {t= lookahead; match(lookahead);
    term(); continue;}
    default: return;
}}
```

Recursive Descent Parsing - Contd

```
term()
{  switch (lookahead)
{ case NUM: match(NUM); break;
   default: error("syntax error");
} }
match( int t)
{ if (lookahead==t) lookahead = lexan();
  else error("syntax error")'
}
```

Idea

You write a function for each of the left side. In the function, you try to call a function which matches each of the symbol in the right hand side.

The basic recursive descent parsing has to back track if the right side of a production does not match.

To avoid backtracking, there are predictive parsers. These are also known as LL(k), where k is the number of symbols that one can look ahead.

Predictive Parsers

Given: An input string and a parsing table M[A,alpha]. Parsing table is a two dimensional array where the rows are the nonterminals and the columns are the terminal symbols. The input string is terminated by $ and initially the stack contains a $ followed by the start symbol.

Algorithm:

1. Set ip to point to the first symbol of the input string w$.

Algorithm Contd

2. REPEAT

a. let X be the top stack symbol and let a be the symbol pointed by ip.

b. If X is a terminal or $ then

if x=a then pop X from the stack and increment ip. else error();

c. else /* X is a nonterminal */ if M[X,a]=Y1..YK then pop X from the stack and push YK,..Y1 onto stack. Output X-->Y1..Yk. else error();

UNTIL X=$

How to Construct the Parse Table

We compute two functions First and Follow.

First(A) is the set of terminal symbols that begin with the strings derived from A. If epsilon is derived from A, then epsilon is in First(A).

Follow(A) is the set of terminal symbols that appear immediately to the right of A, in some sentential form derived from the start symbol. If A is the rightmost symbol in some sentential form then $ is in Follow(A).

S==> alpha A a beta, then a is in Follow(A).

Parse Table Construction

Input: Grammar G Output: Parse Table

Algorithm:

1. For each Production A--> alpha do
2. For each terminal a in First(alpha), add A--> alpha in M[A,a].
3. If epsilon is in First(alpha), for each terminal symbol b in Follow(A), add A--> alpha in M[A,b].
4. Make an undefined entry an error.

Comments and Feedback

Please let me know if you have not found a project partner.

We have finished top-down parsing. Please read chapter 4. (4.1-4.4). Next class, we will cover bottom-up parsing